International Conference on Business Process Management - On the Application of Formal Methods to "Process-Aware" Information Systems, June 26-27, 2003, Eindhoven, The Netherlands: 115 - 120

Towards a Library for Process Programming

Guangxin Yang

Bell-Labs Research, Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ 07974 USA
Email: gxyang@research.bell-labs.com

Abstract. Process programming is regarded as a critical approach in many cooperative process related areas including *software engineering*, *workflow management*, *business process management, etc.* Many process models, languages, and corresponding runtime support systems have been developed. We argue that a comprehensive library for process programming is essential for the acceptance, popularity, and success of this new programming paradigm. We define an architecture of such a library and present some mechanisms on how the architecture is implemented in the context of P, a process language and system for developing integrated cooperation applications.

1 Introduction

Process programming is a vital approach in a number of process management related areas, such as workflow management, process-based software engineering, business process redesign and reengineering. It concerns primarily with modeling and describing formally various aspects, e.g. *activities*, *artifacts*, *roles*, *tools*, and *the interrelationship among them*, of an often complex and long duration procedure within which a group of people cooperate with each other to accomplish a certain task [*Curtis et al. 1992*; *Salimifard et al. 2001*]. The description, or a process program, is critical to understand, design, simulate, optimize, and support processes in real world [*Fuggetta 2000*].

There have been numerous process-modeling techniques, among which language-based ones are dominant. For example, in the workflow area, people have been modeling workflow process with various *Petri net*, *directed graph*, *formal grammar*, etc. based formalisms; in process-centered software engineering, more than a dozen process languages [*Ambrola et al. 1997*], including *APPL/A* [*Sutton 1996*], *SPADE-1* [*Bandinelli 1996*], *HFSP* [*Katayama 1989*], *CSPL* [*Chen 1997*], MELMAC [*Gruhn et al. 1992*], Merlin [*Peuschel et al. 1992*], E^3 [*Jaccheri 1998*], EPOS [*Conradi et al 1994*] have been developed; in business process area, representative work includes *Process Handbook* [*Malon et al. 1999*], the GED framework [*Katzenstein et al. 2000*], and *Process Access Library* [*Hart et al. 1992*] for systematic business process modeling and analyzing. These efforts have produced more and more mature understandings of processes and mechanisms to support them.

Process programming is perhaps based on the understanding that "*business processes are software too*", a generalization of the well received proposition

"*software processes are software too*" in the software process community [*Osterweil 1987*]. Though the power of software reuse is self-evident and many have recognized that it is critical to share and reuse process programs among different systems to promote efficiency and interoperability [see some position papers in *Proc. of IEEE International Software Process Workshop, Dijon, France, 1996*], few systematic efforts have been put into developing advanced mechanisms that enable such sharing and reuse, perhaps partly due to the wide variance in the formalisms, capabilities, and semantics of process languages. We believe that research on mechanisms for process reuse will not only be a key enabler to the popularity of process technologies, but also will lead to better understandings of and more flexible support for processes.

The purpose of this paper is to propose an architecture of reusable process libraries in general and to present our implementation of in the context of the P language and system, which is designed for developing cooperative applications [*Yang et al. 2001*], in particular. Our findings include:

- Process inheritance, which bases one process definition on another, should be the primary pattern for process reuse. With explicit process inheritance, process programmers would be able to exploit the power of abstraction-specialization in developing process programs.
- Process libraries are conceptually a superset of software libraries. Though the main purpose of process libraries is to provide reusable process definitions, software components used at different stages during instance enactments should be also included in process libraries and should be able to be easily expanded.
- Process enactment services, such as creating and manipulating process instances, retrieving the runtime information about instances, should be available via process libraries. The reflection ability to access these services from within process programs would greatly increase the flexibility of process modeling.

The rest of this paper is organized as follows. In the coming section, we will propose a process library architecture. The third section details how the architecture is implemented in the context of P. Comparisons with related work are presented in the forth section. Conclusions and plans for future work are outlined in the last section.

2 An Architecture of Process Libraries

Process libraries should contain the components that can be used as building blocks for writing process programs. Obviously, people want to find reusable process definitions from process libraries to build their own process programs. The problem is once they have these definitions, how they can reuse them efficiently. Inheritance has been a very powerful way for software reuse [*Krueger 1992*]. Since cooperation processes can be programmed, introducing inheritance to process programming should boost dramatically the productivity and popularity of this programming paradigm.

At the same time, cooperative processes are a combination of both coordination and computation. Although there are strong trends and evidences that in the design of some cooperation languages people tend to make a separation between these two parts

[*Cortes et al. 1996*], from the point of view of constructing process libraries, we think both parts are of equal importance. This is because coordination, in some sense, relies on computation. For example, people need to use some kind of software tool, e.g. *a word processor*, *a compiler*, or access some resources, e.g. *a database*, *a file*, to get their work done. Therefore, elements for accessing these services should be an integral part of process libraries.

On the other hand, process programs differ from ordinary software programs in that the former is not directly scheduled by operating systems, but by process systems, which provide functions for managing process instances. Process management services of operating systems, such as creating or terminating processes/threads, are available to any program via certain system calls. Unfortunately, instance management services of process systems are not generally available from within process programs. This prevents process programmers from using these services to build more flexible process programs. Therefore, elements that provide accesses to these kinds of computation services should also be included in process libraries.

An architecture of process libraries based on these considerations is depicted in Fig. 1. We identify three critical building blocks, i.e. reusable process definitions, components for accessing computing services both external and internal to process systems. Each part should be easily extensible, though not necessarily in the same way. Since a complex process may involve any computation, a library for that computation may be included as part of process libraries. It is in this sense that we say process libraries are a superset of classical software libraries. The next three subsections will discuss some of the most important issues within each of the three parts. Our solutions to these issues are presented in Section 3.

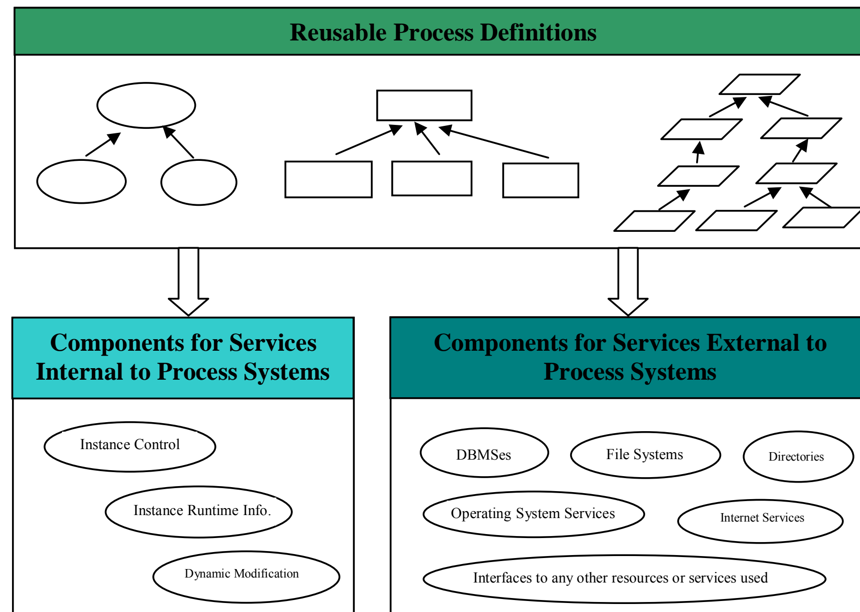

Fig. 1. An architecture of process libraries.

2.1 Reusable Process Definitions

Process inheritance has completely difference semantics, as compared with class inheritance in today's dominate object-oriented programming (OOP). Class inheritance focuses on specializing *object states* and *ways for changing these states*, which are described with member variables and methods, while process inheritance should focuses on specializing *task decomposition and descriptions* and *control and data flows among different tasks*. In the following discussions, if a process inherits another one, we call the former a *subprocess* and the latter its *superprocess*.

Task decomposition can be specialized by adding new task definitions to subprocesses. Task descriptions can be specialized by adding more details and/or overriding it with a new task definition. Control and data flows can be customized by changing the contents exchanged and conditions under which the flows occur. Of course, the specifics depend heavily on the underlying process models of process languages. From the perspective of superprocesses, some mechanisms for hiding these kinds of information from their subprocesses are desirable.

To support the new semantics of inheritance, process systems need to develop new mechanisms both for process description and process enactment. We can borrow some ideas from OOP. For example, we can define a task in a process definition as *private, protected,* or *public* to define explicitly its accessibility to its subprocesses. However these ideas cannot be copied intactly, as the mechanisms for enacting process instances are quite different from those for executing compiled programs. In Fig. 1, process definitions are organized into *trees* according to their inheritance relationship. Here only single inheritance is shown, though it is also reasonable to support multiple inheritance.

2.2 Components for Accessing Services External to Process Systems

A process system cannot stand on its own. Instead, it must talk to the outside world to get information as inputs for instance enactment and to write data to other systems for sharing or recording purposes. Consequently, we must develop appropriate mechanisms so that process programs can access various external resources, e.g. *operating systems*, *file systems*, *databases*, *directories*, *Internet services*, or *even other process systems*. Components for accessing these resources in process libraries are virtually unlimited due to the wide application areas of process technologies.

In designing these mechanisms, we need to take into account a number of factors to address the diversities of external services. We discuss two of them here. The first one is the representation, i.e. the way in which external services are represented in a process system. For example, HFSP has a *tool interface* construct; CSPL has a very similar *tool* unit construct. However, from a more general point of view, we believe that the mechanism should be extended beyond handling only standalone programs to dealing with external computing services in a truly programmable manner by utilizing the application programming interfaces exported by these services, if any.

The second factor is the protocol via which a process system talks to the outside world. In whatever the case, it is the responsibility of the process runtime system to interpret a process program and to communicate with external computing services

when needed, during which a specific protocol may be followed. Since the protocols used by external services may vary greatly from low level protocols like *TCP/IP* to high level standard ones such as *RPC*, *DCOM*, and *CORBA* or proprietary ones such as *Sun Tooltalk* and *DEC FUSE*, the mechanisms should be able to accommodate to these diversity.

2.3 Components for Accessing Services Internal to Process Systems

Process systems implement various process enactment services and hold dynamic information about process instances. These services and information could bridge the somewhat artificial gap that usually exists between the runtime and build-time functions of process systems. We argue that they should be made available to process programs so that process modeling could be more flexible. We believe that by introducing these services and information into process programming, the rigidity of process programs can be significantly decreased and they can be more flexible and adaptive to the ever-changing work conditions.

Firstly, process modeling can be more flexible by using in process programs the runtime information about process instances, which includes *when an instance is created*, *who created it*, *which state it is currently in*, *who are the participants of an activity*, etc.. For example, we can restrict the prospective participants of an activity to be the actual participants of another activity; we can force a program-defined action after a certain time period by using the creating time of an activity or process instance.

Secondly, accessing to instance enactment services gives process programs more control over process instances. Process systems usually treat process instances as its first-class citizens. Though they usually implement all the mechanisms for creating and controlling process instances, few of them support accessing these services from within process programs. Lacking this ability does no harm for toy applications. However, complex real world processes do need this kind of control to manage the creation and enactment of other process instances [*Katayama et al. 1991*].

Thirdly, for process systems that allow dynamic modifications to process definitions, accessing to this functionality from within process programs would make these programs more adaptive to various exception and increase their flexibility. Adaptivity and flexibility have been a hot topic in workflow research [*Klein et al 1998*] and recognized as future direction for software process research [*Fuggetta 2000*]. Existing research has not addressed the possibility and advantage of modifying process programs from within themselves. Our experience has shown that this capability is very useful.

3 Implementations in the P Language and System

We have outlined the architecture of reusable process libraries and discussed some of the important issues in designing these libraries. This section presents how the ideas are implemented in P. We first give a quick introduction to it as the background.

3.1 P: A Brief Overview

The basic construct of a P process is an *activity*, which holds the latest result produced towards the accomplishment of a certain task, e.g. *writing a document or a piece of code*, *filling a form*, or *drawing a picture*, and which contributes, in certain aspect, to the accomplishment of the process. The result can be accessed and modified by cooperators, either exclusively or working together with the applications developed based on the P runtime system.

The P language provides the constructs for modeling various aspects of a process, e.g. its process structure, the rules regarding message exchanges among activities, and the structure and operations of the results held by its activities. More specifically, P has the following four key constructs:

- A *class* provides a full-featured object-oriented language for defining the structures and operations of the results produced in activities and data and control information passed among them.
- An *activity* describes a task that is expected to produce a specific result, which is modeled as a class and is called an activity object (denoted as Õ). An activity can also be a *process*, which we call it a *nested process*.
- A *trigger* is an event-condition-action rule for exchanging both data and control information among activities. Conditions and actions are expressions based on activity objects and other global and predefined objects.
- A *process* describes how a higher-level task should be enacted by grouping a number of activities as a logical unit.

The following code describes a review process. For simplicity, we omit the details of classes *CDocument* and *CReview*. The process defines two activities, i.e. *AProposal* and *AReview*, which produce a *CDocument* and a *CReview* object respectively. The trigger in *AProposal* states that after this activity is finished, the generated document will be visible to activity *AReview*, which will then be started so that a review could be generated. The trigger in *AReview* states that after a review is generated, the summary is fed back to the participants of the design activity via email and if the result is negative, the design activity is restarted so that the design can be modified and resubmitted. Note that in the triggers, the names *AProposal* and *AReview* denotes the activity objects, which are instances of classes *CDocument* and *CReview* respectively.

```
class CDocument    { ... } //details omitted
class CReview      { ... } //details omitted
process PReview startat AProposal
{   //Activity to produce a CDocument object
    public activity AProposal handle CDocument
    {
       trigger TDesignFinished as       //trigger header
          AReview.activity.Resume()     //action
          after submit activity;        //event
    }
    //activity to generate an CReview object
    protected activity AReview handle CReview
    {
```

```
        trigger TReivewFinished as
           email.SendMessage(AProposal.activity.GetParticipants(),
                        "Review Result for" + AProposal.GetName(),
                        AReview.GetReviewSummary() ), //action 1
           AReview.Approved() ? AProposal.SetApproval():
                        AProposal.activity.Resume()   //action 2
           after submit activity;
     }
}
```

For each process instance, the P runtime system maintains some structures to record the instance's runtime state and to execute triggers. One of them is the process object, an instance of the process'es supporting class, which contains two member variables for each activity, one for the activity object and the other for the activity instance. The first member variable is for the process instance. For example, for the process above, the definition of its supporting class is:

```
class PReview_class4process
{  //supporting class for process PReview
   public      CProcessInstance  process; //the process instance
   public      CDocument         AProposal; //for activity object
   public      CActivityInstance AProposal_activity;//activity inst
   protected   CReview           AReview; //for activity object
   protected   CActivityInstance AReview_activity;//activity inst
}
```

CProcessInstance and *CActivityInstance* are predefined classes for process and activity instances. We will describe them in more detail later in Section 3.4. The P runtime system allocates a process object, denoted as Θ, when an instance is initiated and sets the values for its fields when activities are created. To execute a trigger, the runtime system first adds to the supporting class the binary form of a temporary method, which contains a *return* statement, "**return** *ConditionExpr*;", where *ConditionExpr* is the trigger condition, executes it and gets the return value. If the return value is *true*, another temporary method that contains an expression statement, "*ActionExpr;*", where *ActionExpr* is the action expression, is added and executed.

3.2 Process Inheritance

In P, process inheritance is a mechanism that allows all definitions, i.e. *activities* and *local classes* and *processes*, of one process, which is called a *superprocess*, to be part of those of another, which is called a *subprocess*. In the subprocess, these definitions can be used in a way as if they were defined in the subprocess itself. At the same time, the subprocess can define new definitions and/or to customize some definitions in its superprocess/es to fulfill its own specific needs. In this way, a process definition can be reused without losing the capability to be customized. Currently, we support *single inheritance*, which means that a process definition can have only one direct superprocess.

Accordingly, the supporting class for the subprocess is a subclass of the supporting class for the superprocess. However, since the super class already has one member variable of class *CProcessInstance* for the process instance, the subclass will no longer has such a variable. Instead, the first item in the memory block allocated for the subclass now contains the identifier of the memory block allocated for the superclass. In this way, the memory blocks for process objects are chained together. For example, suppose *PPaperReview* is a subprocess of *PReview*. The structure of the memory block of the process object for an instance of *PPaperReview* is shown in Fig.2, where a black cell stores a value converted from the pointer to either an activity or a process instance object (a C++ object in the memory space of the runtime system). Note that the first item of $\Theta_{PReview}$ now stores the converted pointer to process instance for *PPaperReview* (the conversion is done by changing the highest bit from 0 to 1, see Section 3.3 for details.)

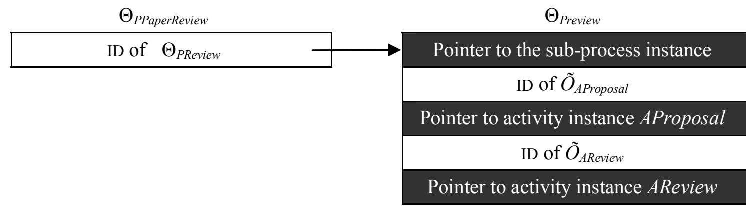

Fig. 2. Memory structure for a process object

3.2.1 Customization

An important purpose of using inheritance is to customize some aspects of superprocesses so that the process definitions can be more appropriate for a more specific situation. In P, we can customize a superprocess in a number of ways, e.g. *adding new activities, overriding or extending existing activities*. A critical issue is to organize the supporting structures for a subprocess so that all definitions in its superprocesses would function properly while instances of the subprocess are enacted.

3.2.1.1 Adding New Activities

A new activity, either a simple one or a nested process, can be added to a subprocess to handle a task that is not described in the superprocesses. By doing this, we customize the task decomposition of a process. The new activity can communicate with activities of superprocesses and of the subprocess itself. For each new activity in the subprocess, we need to add to its supporting class two new member variables and initialize the corresponding fields in the process object with appropriate values when the new activity is instantiated. If the new activity is a simple one, the first variable represents the activity object and the second variable represents the activity instance object. If the new activity is a nested process, the first variable represents the process object for the nested process and the second variable represents the process instance object.

The member variables in the subclass are totally indexed. Index of the first variable in a subclass equals the index of the last variable of its direct superclass plus one. For

example, in Fig. 2, the index of *process*, the first variable of $\Theta_{PReview}$ is 1. If we add a new activity in PPaperReview, the index of the first variable for this activity would be 6. In this way, any expression, no matter whether it is defined in a superprocess or in the subprocess, can be evaluated correctly with the process object for an instance of the subprocess.

3.2.1.2 Overriding or Extending Existing Activities

In some cases, we may just want to customize some aspects of an activity of a superprocess. For example, we can change the class of the activity object so that the object produced in this activity can be more specific; we can disable or enable some of its triggers to turn off or on the communication channels between this activity and others; we can add new triggers to the activity to change the way it communicate with other activities; we can change its participant specification; we can even replace the activity definition with a new one.

As a simple example, the code below defines a subprocess, *PPaperReview*, of *PReview* defined in Section 3.1. We first create a new class *CPaper*, which inherits *CDocument*. Then we customize the design activity by changing the class of the activity object to *CPaper*. The review activity is customized by specifying the participants to be a set of reviewers whose expertise matches the keywords of the submitted paper and by adding a new trigger that will send a message to the perspective reviewers every time the review activity is resumed. As indicated by the keyword *extending*, the two activity definitions in the subprocess customize the activities in the superprocess by extending them. Therefore, the trigger definitions in the superprocess will still be in effect.

```
class CPaper extend CDocument  { … } //details omitted
process PPaperReview extend PReview
{   //make the activity object more specific
    extending activity AProposal handle CPaper
    {
    }
    //add participant specification and a new trigger
    extending activity AReview handle CReview
    {
       users reviewerdb.GetReviewers(AProposal.GetKeywords());
       trigger TNotifyReviewers as      //trigger header
          email.SendMessage(
             reviewerdb.GetReviewers(AProposal.GetKeywords()),
             "Paper for your review",
             AProposal.GetPaperURL())      //action
          after resume activity;        //event
    }
}
```

Overriding or extending an activity does not introduce a new activity (task) to the process. This means that for that activity, only one activity object and only one activity instance object will be created. For example, *PPaperReview* has only two activities, *AProposal*, which produces a *CPaper* object and *AReview*, which produces a *AReview* object. The P runtime system properly reorganizes the supporting

structures for a process instance to reflect this fact. For example, for an instance of process *PPaperReview*, after its two activities are both activated, the state of its process object is shown in Fig. 3. Note that $\Theta_{PPaperReview}$ has four variables for the two extending activities. However, the value of each variable of $\Theta_{PReview}$ will be set to that of the corresponding variable of $\Theta_{PPaperReview.}$ In this way, the corresponding variables represent the same entity (an activity object or an activity instance object).

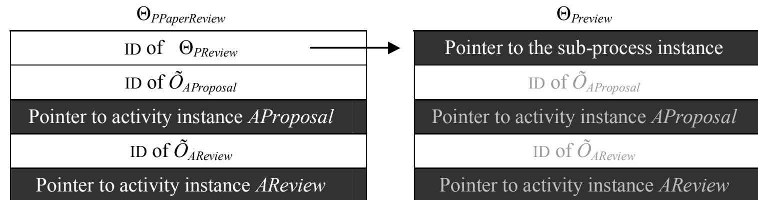

Fig. 3. The process object of process *PPaperReview*

Overriding and extending differ in how they affect the runtime system in searching the triggers in activities with the same name in the process inheritance hierarchy. Starting from the leaf node, the searching ends at the first overriding activity. To ensure that trigger actions and conditions referring the activity can be correctly executed or evaluated, we require that the class type of the overriding or extending activity is the same as or a subclass of that of the activity being overridden or extended. For an activity that is a nested process, we only allow to override it by defining a new nested-process activity whose process type is a subprocess of that of the existing one.

3.2.2 Activity Accessibility

Process inheritance and process nesting raise the issue of activity accessibility, i.e. whether a subprocess or a nesting process can communicate with the activities in its superprocesses or the nested process. For example, a process programmer may want an activity to be used for a completely internal purpose and don't want any activity in a subprocess or a nesting process to communicate with it. P has a mechanism that is very similar to how some object-oriented languages restrict the visibility of class variables and methods. We define three accessibility modifiers, i.e. *private*, *protected*, and *public*. A private activity is only accessible in the triggers, participants, etc. of the activities in the same process definition. A protected activity can be accessed by the activities in the same process and its subprocesses. A public activity of a nested process can be accessed from within a nesting process. The P compiler checks the accessibility of activities when a P process program is compiled. It reports a compiler error if the rules are violated.

3.3 Native Methods/Objects for External Resources

P is an interpretive language. The P interpreter manages the allocation, release, and referencing of P objects used for various purposes during the enactment of instances and interprets the binary process codes compiled from P source programs. P also supports a mechanism that allows the body of a method of a P class to be a native function residing in a dynamical link library (DLL) on Win32 platforms (our current implementation). We call it a *native method*. When a native method is called, the P interpreter calls the corresponding native function with all actual parameters. The return value will be copied to P's storage space. The native function can be written in any language. Our mechanism is very similar to Java's Native Interface (JNI).

As a more advanced form of the *native method* mechanism, another mechanism allows us to introduce a C++ object running in the memory space of the P runtime system into the object space of the P interpreter as a P object, which we call it a *native object*. The P interpreter will direct a call to a method of a native object to the corresponding method of the C++ object. For this purpose, we need to write a P class definition that services as an interface description of the C++ object. (The class *CProcessInstance* in Section 3.4.1 is an example.) The identifier of a native object is a value converted from the pointer of the corresponding C++ object by changing its highest bit from 0 to 1. (On Win32 platforms, a pointer in the user space is within range $0 \sim 2^{31}-1$. Therefore, the identifier of a native object will be in range $2^{31} \sim 2^{32}-1$. The identifier of an ordinary P object allocated in P interpreter falls in range $0 \sim 2^{31}-1$. Therefore, the P interpreter is able to tell whether an identifier represents an ordinary P object or a native object.)

The native method/object mechanisms offer a fully extensible and programmable way for accessing any external services from within P process programs. They are consistent with the object-oriented pattern for programming P classes. For a certain resources, all we need to do is to define as a P class an interface for it and implement the native functions or objects. Based on this mechanism, we have successfully developed some P classes, e.g. *CTime*, *CMath*, *CIO*, *CFileSystem, CEMailAgent*, and *CDatabase*, and the related native codes for *time conversion*, *mathematical calculation*, *input/output*, *accessing files, sending emails*, and *accessing a relational database*, with several hours work.

3.3.1 Global Objects

In some applications, some resources may be shared by many process instances. In these cases, we can install dynamically in a P runtime system objects representing these resources as global objects, which are accessible from within any process instance enacted in that runtime system. The P runtime system has the programming interface for dynamically install and uninstall global objects. A global object can be either a native object, or a normal P object whose definitions are given in the P language.

The processes *PReview* and *PPaperReview* contain examples on how to use global objects. There we use two global objects, *email,* which is an instance of *CEMailAgent* for sending Internet email messages, and *reviewerdb*, which is an instance a subclass of *CDatabase*. These objects should be properly installed when instances of *PReview* and *PPaperReview* are enacted.

3.4 Predefined Objects for Internal Services

The P runtime system allocates in its memory space a C++ object that records the runtime information of the process instance. It also allocates for each activity instance a C++ object representing the activity instance. For reflection purpose, it installs these C++ objects as native objects in the P interpreter. In a process program, these objects can be accessed via two predefined identifiers, i.e. *process, activity*, in the same way as ordinary P objects are accessed.

3.4.1 process

process represents a process instance. It is a native object, which is mapped to a C++ object maintained for the instance in the P runtime system. *process* is an instance of class *CProcessInstance*, whose methods can be divided into four categories:

- *Retrieving Information* for retrieving certain information, e.g. *process name, display name, state code, start time,* and end time, that is determined at runtime.
- *Manipulating Instance* for changing forcefully, i.e. suspending, resuming, terminate, and aborting, the state of the process instance.
- *Modifying Definition* for modifying dynamically the process definition of this instance so that the instance can be adaptive to changing working setting. Details on this topic are available in a technical memo [*Yang 2002*].
- *Creating New Instance* for creating dynamically a new process instance to coordinate the accomplishment a new task. The newly created instance will be a top-level instance and enacted independently of the current one.

The definition of *CProcessInstance* is given below.

```
class CProcessInstance
{  //the constructor
   public          CProcessInstance(){ }
   //the name of the process definition
   public CString GetProcessName()  { }
   //return the user friendly name
   public CString GetInstanceName() { }
   //return the state of this process instance
   public tiny    GetState()        { }
   //Get the start time of the process instance
   public int     GetStartTime()    { }
   //Get the end time of the process instance
   public int     GetEndTime()      { }
   //abort this process instance
   public boolean Abort()           { }
   //terminate this process instance
   public boolean Terminate()       { }
   //resume this process instance
   public boolean Resume()          { }
   //suspend this process instance
   public boolean Suspend()         { }
   //create a new process instance
   public boolean CreateInstance(CString csProcessName,
```

```
        CString csDisplayName, CString csArguments)  { }
    //modify the definiton of this process
    public boolean ModifyDefinition(
        tiny tAction, CString csObjectName,
        CString csSubObjectName, CString csCode)     { }
}
```

3.4.2 activity

activity represents an activity instance in a process instance. It is also a native object, which is mapped to the corresponding C++ object in the P runtime system. The P interpreter records the value converted from the pointer to the C++ object in the process object for that process instance. *activity* is an instance of class *CActivityInstance*, which has methods for retrieving certain information about the instance and manipulating it. For simplicity, we will not list the code here. Both *PReview* and *PPaperReview* have examples on how *activity* can be used. (P compiler will convert a notation like *AReview.**activity*** into *AReview_**activity***, which is a member in the supporting class for the process.)

4 Comparisons with Related Work

We have outlined the architecture of a library for process programming and presented our implementation in the context of the P language and system. To the best of our knowledge, this is the first attempt to find a systematic solution for building reusable libraries for process programming. Though some ideas such as *inheritance*, *external resource integration*, and *reflection* are by no means new and have been exploited in some form in other work, a comparison with this related work would reveal that our implementation of these ideas are in some sense novel.

Several groups have reported building a process library for sharing and analyzing purposes. [*Gruhn et al. 1998*] presents the mechanisms on how to manage the storage of process definitions in both a relational *DBMS* and the main memory. These mechanisms aim at improved performance. The *Process Handbook* aims at a repository of business process descriptions to help people redesign existing and invent new organizational processes and to share ideas of organizational practices. It uses *inheritance* as a second dimension, in addition to *task decomposition*, to explore the similarities among different processes. The descriptions are made with natural English, thus the mechanisms like these in this paper for process enactment are out of their scope. In an similar effort, the *Process Asset Library* [*Hart et al. 1992*], the goal is to organize all assets, e.g. *generic process architectures*, *cycle models*, *process elements (or subprocesses or steps)*, of business processes into one easily available online database. Again the description, organization, and categorization of business processes are their main focus.

Some process models and languages such as *EPOS*, E^3, *CSPL*, and *GED* [*Katzenstein et al. 2000*] support inheritance. However, the inheritance is limited to describing in an object-oriented fashion various entities, e.g. *artifacts*, *roles*, *tasks*, involved in software processes. We have not yet seen any process language that

supports truly process level inheritance. Thus none of them has ever tried to develop related enactment mechanisms. [*Aalst et al. 2001*] defines formally four different types of inheritance of processes modeled with WF-net, a special kind of PetriNet. Their main purpose is to analyze and tackle problems related to transferring running workflow instances to a new model. We focus on mechanisms for enacting properly processes inherited from other processes.

External tool integration has been a major concern of nearly all process languages in software engineering. For those based on classical languages, such as Ada-based *APPL/A*, Prolog-based *Merlin*, it is possible to use the libraries of the base language directly to access any external resources. Therefore they don't need any special construction. For newly defined languages such as *SPADE-1*, *CSPL*, and *HSPF*, they need to introduce new construction for specifying external tools. For example, *CSPL* has a *tool* construct; an HSPF program can define a *tool* section to define tools used in it; *SPADE-1* uses more powerful tool integration facilities, such as *Sun Tooltalk*, *Microsoft OLE2*, and *DEC FUSE*; and E^3 uses *CORBA*. Our native method/object based mechanism allows external resources to be integrated virtually without any limitation, though similar mechanism has been used in other general-purpose languages like Java.

Accessing to internal services is a kind of reflection, a feature also available in other language like Smalltalk and Java. Reflection in certain limited forms is found in some process languages. For example, *HSPF* has a special data type *status* and two special operations *snap* and *resume* for recording and changing the enactment status of a process; *MELMAC* has a command for modifying a process model fragment immediately before it is enacted; *EPOS* and *SPADE-1* offers basic, reflexive constructs to support the specification of the "process of change" as part of the process itself. P's *process* and *activity* offer a more comprehensive reflection mechanism to use from within process programs all the enactment services provided by the P runtime system. Also it is extensible in the sense that we can easily add new functions. An added advantage is that they can be used in a consistent object-oriented manner.

5 Conclusions and Future Work

Process programming as a new programming paradigm is believed to be critical in many tightly related areas, e.g. software engineering, workflow management, and business management. We believe that a library with reusable elements for programming cooperative processes flexibly is essential for the acceptance, popularity, and success of this new paradigm. We defined an architecture for such a library and present an implementation in the context of a process programming language named P. We demonstrated that process inheritance can be an efficient vehicle for process reuse and developed the mechanisms that implement the inheritance semantics.

A public release of the P runtime system that contains all the work discussed here is available at http://blrc.edu.cn/research/p/. The system has been successfully used as the programming environment for a graduate level course on CSCW in Tsinghua

University and as the software platform for several projects in e-commerce and workflow management. Recently, we have developed a process program for the ISPW6 reference problem [*Kellner et al. 1991*]. These experiences show that the library concepts discussed here work pretty well.

However, there is still a long way to go. Making our library much richer, e.g. by adding more reusable process definitions into our library, by developing special-purposed libraries for different application areas, is in our near-term plan. The architecture and mechanisms need to be further explored and validated in a wider range of practical applications, with which we may learn something to improve and enhance our concepts and mechanisms.

A long-term work we are currently considering is whether it is necessary to define a standard binary format of process programs, and if the answer is yes, what format should we define. We believe that a standard format will promote significantly process sharing and reusing. This attempt conforms to NIST's goal on *Process Specification Language* [*http://ats.nist.gov/psl/*] and WfMC's on developing a common process interchange standard based on XML [*WfMC 2002*]. However, our effort differs from these in that it tries to achieve sharing from a much lower level.

The work itself is very challenging due to many factors, both technical and non-technical. For example, process programs are *executed* by process systems, which may adopts different process meta-models. It may be difficult, if not impossible, to map the concepts of one meta-model to the ones of another. However, if this can be done, all process systems will have a common ground to play on. The libraries for process programming may then be developed with different process languages.

Acknowledgements

The work presented here is based on the author's PhD work, which was supervised by Prof. Shi, Meilin at Tsinghua University and supported by several grants from China NSF and 863 Plan. The author is grateful for all the comments and suggestions from the students who ever used P for their coursework. The author wants to thank Alfred V. Aho and the anonymous reviewers for their insightful comments and suggestions.

References

Aalst, W. M. P. and Basten, T. Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 2001, 270(12):125-203

Ambriola, V., Conradi, R., and Fuggetta, A. Assessing process-centered software engineering environments. *ACM TOSEM*, 1997, 6(3):283-328

Bandinelli, S., Nitto, E. D., and Fuggetta, A. Supporting cooperation in the SPADE-1 environment. *IEEE TOSE*, 1996, 22(12):841-865

Chen, J. Y. CSPL:an Ada95-like, unix-based process environment. *IEEE TOSE*, 1997, 23(3):171-184

Conradi, R., Hsgaseth, M., Larsen, J.-O., *et al.* EPOS: Object-oriented cooperative process modeling. In: *Software Process Modeling and Technology*, Finkelstein, A., Kramer, J. and Nuseibeh, B. *Eds*. Research Studies Press, London, U.K., 1994

Cortes, M. and Mishra, P. DCWPL: A Programming Language for Describing Collaborative Work. In: *Proc of ACM CSCW*, Cambridge, 1996, 21-29

Curtis, B., Kellner, M. I., and Over, J. Process modeling. *CACM*, 1992, 35(9):75-90

Fuggetta, A. Software process:a roadmap. In: *Proc of ACM/IEEE ICSE*, Limerick, 2000, 27-34

Gruhn, V. and Schneider, M. Workflow management based on process model repositories. In: *Proc of ACM/IEEE ICSE*, Kyoto, 1998, 379-388

Gruhn,V. and Jegelka, R. An evaluation of FUNSOFT nets. In: *Proc of the 2nd European Workshop on Software Process Technology*, 1992, LNCS. Springer-Verlag, New York.

Hart, H., Doland, J., Drake, D., *et al*. STARS process concepts summary. In: *Proc of ACM Conf on TRI-Ada*, Orlando, 1992 , 570-594

Jaccheri, M. L., Picco, G. P., and Lago, P. Eliciting software process models with the E^3 language. *ACM TOSEM*, 1998, 7(4):368-410

Katayama, T. A hierarchical and functional software process description and its enaction. In: *Proc of ACM ICSE*, Pittsburgh, 1989, 243-252

Katayama, T. and Motizuki, S. What has been learned from applying a formal process model to a real process. In: *Proc of IEEE ISPW*, Washington D.C., 1991, 79-81

Katzenstein, G. and Lerch, F. J. Beneath the surface of organizational processes: a social representation framework for business process redesign. *ACM **TOIS***, 2000, 18(4):383-422

Kellner, M. I., Feiler, P. H., Finkelstein, A., *et al*. ISPW-6 software process example. In: *Proc of IEEE ISPW*, Redondo Beach, 1991, 176-186

Klein, M., Dellarocas, C., and Bernstein, A. *eds*. *Proc of ACM CSCW Workshop on Adaptive Workflow Systems*, Seattle, 1998. (http://ccs.mit.edu/klein/cscw98/.)

Krueger, C. W. Software reuse. *ACM Computing Surveys*, 1992, 24(2):131-183

Malon, T. W., Crowston, K., Lee J., *et al*. Tools for inventing organizations: toward a handbook of organizational processes. *Management Science*, 1999, 45(3):425-443

Osterweil, L. Software processes are software too. In: *Proc of ACM/IEEE ICSE*, Monterey, 1987, 2-13

Peuschel, B. and Schafer, W. Concepts and implementation of a rule-based process engine. In: *Proc of ACM/IEEE ICSE*, Melbourne(AU), 1992, 262-279

Salimifard, K. and Wright, M. Petri net-based modelling of workflow systems: an overview. *European Journal of Operational Research*, 2001, 134, 664-676

Sutton, S. M., Jr., Heimbigner, D., and Osterweil, L. J. APPL/A: a language for software process programming. *ACM TOSEM*, 1996, 4(3):221-286

WfMC. Workflow process definition interface – XML process definition language. *WFMC-TC-1025*, 2002. (http://www.wfmc.org/.)

Yang, G. and Shi, M. *Cova*: a programming language for cooperative applications. *Science in China, Series F*, 2001, 44(1):73-80

Yang, G. Inheritance-based dynamic process modifications. *Bell-Labs Technical Memo*, Nov. 2002